A gallery of AD systems

UCSD CSE 291 Differentiable Programming
Tzu-Mao Li
Today: look at a bunch of AD systems
CppAD: a tracing based AD system for C++
#include <cppad/cppad.hpp>  // single header library

// ...
using CppAD::AD;
vector<AD<double>> ax(2);
ax[0] = 0; ax[1] = 1;

// declare independent variables and starting recording
CppAD::Independent(ax);

vector<AD<double>> ay(1);
ay[0] = ax[0] * ax[0] * ax[1];

// create f: x -> y and stop tape recording
CppAD::ADFun<double> f(ax, ay);

vector<double> dx(2), dy(1);
dx[0] = 1.0; dx[1] = 0.0;
dy = f.Forward(1 /* 1st derivative */, dx);
// dy[0] now contains the derivative
Discussion: what are pros and cons of CppAD?

```cpp
#include <cppad/cppad.hpp>  // single header library

// ...
using CppAD::AD;
vector<AD<double>> ax(2);
ax[0] = 0; ax[1] = 1;

// declare independent variables and starting recording
CppAD::Independent(ax);

vector<AD<double>> ay(1);
ay[0] = ax[0] * ax[0] * ax[1];

// create f: x -> y and stop tape recording
CppAD::ADFun<double> f(ax, ay);

vector<double> dx(2), dy(1);
dx[0] = 1.0; dx[1] = 0.0;
dy = f.Forward(1 /* 1st derivative */, dx);
// dy[0] now contains the derivative
```
CppAD Codegen
# CppAD Codegen example

```cpp
#include <cppad/cg.hpp> // single header lib!
// ...
using CppAD::AD;
using CGD = CG<double>
using ADCG = AD<CGD>;

vector<ADCG> ax(2);
ax[0] = 0.; ax[1] = 1.;
CppAD::Independent(ax);
vector<ADCG> ay(1);
ay[0] = ax[0] * ax[0] * ax[1];
CppAD::ADFun<CGD> f(ax, ay);

// generate C code
CodeHandler<double> handler;
vector<CGD> indVars(2);
handler.makeVariables(indVars);

// record CppAD's trace
vector<CGD> jac = f.Jacobian(x);
LanguageC<double> langC("double");
LangCDefaultVariableNameGenerator<double> nameGen;

std::ostringstream code;
handler.generateCode(code, langC, jac, nameGen);
std::cout << code.str();
```
**Discussion: what are pros and cons of CppAD Codegen?**

```cpp
#include <cppad/cg.hpp> // single header lib!

using CppAD::AD;
using CGD = CG<double>
using ADCG = AD<CGD>;

vector<ADCG> ax(2);
ax[0] = 0.; ax[1] = 1.;
CppAD::Independent(ax);
vector<ADCG> ay(1);
ay[0] = ax[0] * ax[0] * ax[1];
CppAD::ADFun<CGD> f(ax, ay);

// generate C code
CodeHandler<double> handler;
vector<CGD> indVars(2);
handler.makeVariables(indVars);

// record CppAD's trace
vector<CGD> jac = f.Jacobian(x);

LanguageC<double> langC("double");
LangCDefaultVariableNameGenerator<double> nameGen;

std::ostringstream code;
handler.generateCode(code, langC, jac, nameGen);
std::cout << code.str();
```
ADOL-C

- basically CppAD, but with a more C-like interface
- written by famous AD people (Walther and Griewank), often used for testing SOTA (scalar) AD algorithms
Adept

• similar to CppAD, with two clever tricks for speed up
• store partial derivatives in tape, instead of actual instructions
• use expression templates to do as much work at compile time as possible
Adept

Fast Reverse-Mode Automatic Differentiation using Expression Templates in C++

ROBIN J. HOGAN, University of Reading

Gradient-based optimization problems are encountered in many fields, but the associated task of differentiating large computer algorithms can be formidable. The operator-overloading approach to performing reverse-mode automatic differentiation is the most convenient for the user but current implementations are typically 10-35 times slower than the original algorithm. In this paper a fast new operator-overloading method is presented that uses the expression template programming technique in C++ to provide a compile-time representation of each mathematical expression as a computational graph that can be efficiently traversed in either direction. Benchmarking with four different numerical algorithms shows this approach to be 2.6–9 times faster than current operator-overloading libraries, and 1.3–7.7 times more efficient in memory usage. It is typically less than 4 times the computational cost of the original algorithm, although poorer performance is found for all libraries in the case of simple loops containing no mathematical functions. An implementation is freely available in the Adept C++ software library.

Categories and Subject Descriptors: G.1.4 [Numerical Analysis]: Quadrature and Numerical Differentiation—Automatic differentiation; G.1.6 [Numerical Analysis]: Optimization—Gradient measures

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Adjoint code, quasi-Newton, template metaprogramming, Jacobian matrix

ACM Reference Format:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>LW</th>
<th>Toon</th>
<th>PVC</th>
<th>TDTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time of original algorithm</td>
<td>0.60 ms</td>
<td>13.1 ms</td>
<td>0.013 ms</td>
<td>0.85 ms</td>
</tr>
</tbody>
</table>

Table I. Speed Comparison for Reverse-Mode Differentiation Applied to the Advection Schemes of Lax and Wendroff [1960] (LW) and Toon et al. [1988] (Toon) using 2000 Timesteps, and to the Radiative Transfer Models based on the Photon Variance-Covariance (PVC) and Time-Dependent Two Stream (TDTS) Methods for Profiles of Atmospheric Variables at N = 50 Height Intervals, Taken from Version 1.2.10 of the Multiscatter Package
Enzyme

• directly differentiate LLVM and MLIR code!!
• very impressive set of features (OpenMP/MPI/GPU backends, support differentiating almost all C code)
• actively being developed, with some Google developers
• probably my goto AD tool if I have some C/C++ code needs to be differentiated
Key idea of Enzyme

Existing Automatic Differentiation Pipelines

https://indico.cern.ch/event/1145124/contributions/4994088/attachments/2508821/4311554/enzyme-mode.pdf
Key idea of Enzyme

Case Study: Vector Normalization

```c
//Compute magnitude in O(n)
double mag(double[] x);

//Compute norm in O(n^2)
void norm(double[] out, double[] in) {
    for (int i=0; i<n; i++) {
        out[i] = in[i] / mag(in);
    }
}
```

https://indico.cern.ch/event/1145124/contributions/4994088/attachments/2508821/4311554/enzyme-mode.pdf
Key idea of Enzyme

Case Study: Vector Normalization

```c
//Compute magnitude in O(n)
double mag(double[] x);

//Compute norm in O(n)
void norm(double[] out, double[] in) {
    double res = mag(in);
    for (int i=0; i<n; i++) {
        out[i] = in[i] / res;
    }
}
```

https://indico.cern.ch/event/1145124/contributions/4994088/attachments/2508821/4311554/enzyme-mode.pdf
Key idea of Enzyme

Optimization & Automatic Differentiation

\[ O(n^2) \]
for i=0..n {
  out[i] /= mag(in)
}

Optimize

\[ O(n) \]
res = mag(in)
for i=0..n {
  out[i] /= res
}

AD

\[ O(n) \]
d_res = 0.0
for i=n..0 {
  d_res += d_out[i]...
dmag(d_in, d_res)
}

\[ O(n^2) \]
for i=0..n {
  out[i] /= mag(in)
}

AD

\[ O(n^2) \]
for i=n..0 {
  d_res = d_out[i]...
dmag(d_in, d_res)
}

Optimize

\[ O(n^2) \]
for i=n..0 {
  d_res = d_out[i]...
dmag(d_in, d_res)
}

Source: https://indico.cern.ch/event/1145124/contributions/4994088/attachments/2508821/4311554/enzyme-mode.pdf
Key idea of Enzyme

Performing AD at low-level lets us work on **optimized** code!

https://indico.cern.ch/event/1145124/contributions/4994088/attachments/2508821/4311554/enzyme-mode.pdf
• a DSL solution focuses on dense array computations, especially image processing
• used in Photoshop, Youtube, Google pixels, Qualcomm Hexagon, and more
• heavily influenced DL systems like TVM
• part of my PhD work was to extend Halide to handle differentiation
A very cool talk from Alex Reinking

Halide: A Language for Fast, Portable Computation on Images and Tensors
Alex Reinking

https://www.youtube.com/watch?v=1ir_nEfKQ7A
Motivation: existing automatic differentiation systems are limited
depth learning framework:
  too coarse-grained, inflexible
automatic differentiation libraries:
  too general purpose, inefficient
Developing custom operators is tedious.

derive the gradient (manually)
implement
debug
repeat

CUDA 308 lines  code from Gharbi et al. 2017
Deep learning frameworks are limited

- coarse-grained operators
- no control over performance

**CUDA**
- 308 lines
- 430 ms (1M pix)
- 2270 ms (4M pix)

**PyTorch**
- 42 lines
- 1440 ms (1M pix)
- out of memory (4M pix)
Our solution is automatic, flexible, and efficient.

CUDA  308 lines
430 ms (1M pix)
2270 ms (4M pix)

PyTorch  42 lines
1440 ms (1M pix)
out of memory (4M pix)

Ours  24 lines
64 ms (1M pix)
165 ms (4M pix)
Halide’s key idea: separation of algorithm and schedule

high-level algorithm:
  gamma correction

Func f;
f(x, y) = pow(im(x, y), g);
Halide's key idea: separation of algorithm and schedule

high-level algorithm: gamma correction

```c
Func f;
f(x, y) = pow(im(x, y), g);
```

low-level schedule: order and storage

```c
f.vectorize(x, 4)
```

```c
f.parallel(y, 2)
```
An example: image blurring

(a) Clean C++: 9.94 ms per megapixel

```cpp
void blur(const Image &in, Image &blurred) {
    Image tmp(in.width(), in.height());

    for (int y = 0; y < in.height(); y++)
        for (int x = 0; x < in.width(); x++)
            tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;

    for (int y = 0; y < in.height(); y++)
        for (int x = 0; x < in.width(); x++)
            blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;
}
```

(b) Fast C++ (for x86): 0.90 ms per megapixel

```cpp
void fast_blur(const Image &in, Image &blurred) {
    #pragma omp parallel for
    for (int yTile = 0; yTile < in.height(); yTile += 32) {
        __m128i one_third = _mm_set1_epi16(21846);
        __m128i a, b, c, sum, avg;
        __m128i tmp16[32/2];
        for (int xTile = 0; xTile < in.width(); xTile += 256) {
            __m128i *tmpPtr = tmp;
            for (int y = -1; y < 32+1; y++) {
                const uint16_t *inPtr = &(in(xTile, yTile+y));
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_loadu_si128((__m128i*)inPtr--);
                    b = _mm_loadu_si128((__m128i*)inPtr--);
                    c = _mm_loadu_si128((__m128i*)inPtr--);
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_add_epi16(sum, one_third);
                    _mm_storeu_si128(tmpPtr++, avg);
                    inPtr += 8;
                }
            }
            tmpPtr = tmp;
            for (int y = 0; y < 32; y++) {
                __m128i *outPtr = (__m128i*)(&blurred(xTile, yTile+y));
                for (int x = 0; x < 256; x += 8) {
                    a = _mm_loadu_si128(tmpPtr++(2+256)/8);
                    b = _mm_loadu_si128(tmpPtr++(2+256)/8);
                    c = _mm_loadu_si128(tmpPtr++(2+256)/8);
                    sum = _mm_add_epi16(_mm_add_epi16(a, b), c);
                    avg = _mm_add_epi16(sum, one_third);
                    _mm_storeu_si128(outPtr++, avg);
                }
            }
        }
    }
}
```

(c) Halide: 0.90 ms per megapixel

```cpp
Func halide_blur(Func in) {
    Func tmp, blurred;
    Var x, y, xi, yi;

    // The algorithm
    tmp(x, y) = (in(x-1, y) + in(x, y) + in(x+1, y))/3;
    blurred(x, y) = (tmp(x, y-1) + tmp(x, y) + tmp(x, y+1))/3;

    // The schedule
    blurred.tile(x, y, xi, yi, 256, 32)
        .vectorize(xi, 8).parallel(y);
    tmp.chunk(x).vectorize(x, 8);

    return blurred;
}
```
We can differentiate through general programs

piecewise-linear tone mapping
We can differentiate through general programs

\[
f = \text{floor}(\text{in}(x, y))
\]
\[
c = \text{ceil}(\text{in}(x, y))
\]

piecewise-linear tone mapping
We can differentiate through general programs

\[ f = \text{floor}(\text{in}(x, y)) \]
\[ c = \text{ceil}(\text{in}(x, y)) \]
\[ w = \text{in}(x, y) - f \]
\[ \text{out}(x, y) = \text{lut}(f) \times (1 - w) + \text{lut}(c) \times w \]
We can differentiate through general programs

// Differentiable soft histogram
f = floor(in(r.x, r.y))
c = ceil(in(r.x, r.y))
w = in(r.x, r.y) - f
hist(f) += (1 - w)
hist(c) += w
A few lines of code generates gradients

```cpp
// loss depends on im and param
Func loss;
loss() = ...;

auto d_loss_d = propagate_adjoints(loss);
Func d_im = d_loss_d(im);
Func d_param = d_loss_d(param);
```

there’s also a PyTorch interface!
Long pipelines may require large memory

draw \rightarrow \text{demosaic} \rightarrow \text{white balance} \rightarrow \text{gamma} \rightarrow \ldots

d_{\text{demosaic}} \leftarrow d_{\text{wb}} \leftarrow d_{\text{gamma}}
Long pipelines may require large memory

gamma = pow(wb, g)
d_wb = d_gamma * pow(wb, g-1) * wb
Halide lets us trade-off memory/recompute

aka checkpointing

demosaic.compute_root() // Cache

gamma.compute_inline() // Recompute
Key idea: differentiate the algorithm, separately schedule primal code and derivatives

the exact opposite of Enzyme!! who is right?

1D array summation

\[ \text{out}() += \text{in}(r.x) \]

differentiate

\[ \text{d}_\text{in}(x) = \text{d}_\text{out}() \]

parallel reduction

\[ \text{interm} = \text{out}.\text{update}() \]
\[ \quad .\text{split}(r.x, \text{ro}, \text{ri}, 32) \]
\[ \quad .\text{rfactor}([\text{ro}, x]); \]
\[ \quad \text{interm}.\text{compute_root}() \]
\[ \quad .\text{update}().\text{parallel}(\text{ro}) \]
\[ \quad .\text{vectorize}(16); \]

schedule

\[ \text{d}_\text{in}.\text{split}(x, \text{xo}, \text{xi}, 32) \]
\[ \quad .\text{parallel}(\text{xo}) \]
\[ \quad .\text{vectorize}(\text{xi}) \]
Improving image processing algorithms by updating their parameters

(a) AHD

(b) ours, 8 5x5 filters

(c) ground truth

19.6 dB

24.7 dB
Slang

disclaimer: I worked on a paper about differentiating Slang

- focuses on GPU shading languages
- heavily pushed by NVIDIA right now!
- Source Engine 2 just switched to Slang recently
Motivation:
we want to reuse heavily-engineered rendering code

an ultra-fast renderer **Falcor**
written by NVIDIA engineers

> 252,000 lines C++ CPU host code
> 5,000 lines GPU shader code

3 billion triangles + 3.5M hair segments, rendered at 1920x1080 with 30 frames per second
Goal: compute derivatives with minimal change to the existing codebase
The Falc0r renderer is implemented in Slang

Slang
- is a SIMD language based on HLSL
- plays well with graphics hardware instructions (ray tracing, rasterization, texture units, tensor cores, …)
- supports many backends: HLSL, GLSL, CUDA, OptiX, SPIR-V, C, …
- supports generics and interfaces for static and dynamic polymorphism
- battle-tested, maintained by a whole team in NVIDIA
The Falcor renderer is implemented in Slang

Slang
- is a SIMD language based on HLSL
- plays well with graphics hardware instructions (ray tracing, rasterization, texture units, tensor cores, …)
- supports many backends: HLSL, GLSL, CUDA, OptiX, SPIR-V, C, ...
- supports generics and interfaces for static and dynamic polymorphism
- battle-tested, maintained by a whole team in NVIDIA

```c
StructuredBuffer<float> buffer0;
StructuredBuffer<float> buffer1;
RWStructuredBuffer<float> result;

[shader("compute")]
[numthreads(1,1,1)]
void computeMain(uint3 threadId : SV_DispatchThreadID) {
    uint index = threadId.x;
    result[index] = buffer0[index] + buffer1[index];
}
```
Key feature in Slang:
unification of specialization and dynamic dispatch

similar to Haskell’s type classes, Rust’s type trait, and C# interfaces

void myShader(...) {
  <geom>.compute(...);
  <material>.compute(...);
  <light>.compute(...);
}
Key feature in Slang:
unification of specialization and dynamic dispatch

similar to Haskell’s type classes, Rust’s type trait, and C# interfaces

void myShader(…) {
  <geom>.compute(…);
  <material>.compute(…);
  <light>.compute(…);
}

in practice, we often want to specialize to a particular
graphical geometry/material/light in our shader,
removing virtual tables/conditions
Key feature in Slang: unification of specialization and dynamic dispatch

similar to Haskell’s type classes, Rust’s type trait, and C# interfaces

```
interface IMaterial {
    Color compute(...);
}

struct Matte : IMaterial {
    Color compute(...) {...}
}

void myShader(IMaterial m, ...) {
    m.compute(...)
    ...
}
```

if `myShader` is invoked with a static known material type, then the compiler would specialize the method using the concrete type, otherwise it inserts a `switch` statement
Goal of SLANG.D: automatically differentiate Slang code

```c
// Annotate methods to signal differentiability
[Differentiable]
float square(float x) {
  return x * x;
}

float3 main() {
  DifferentialPair<float> dpx(4.f, 0.f);
  // Call the derivative of the 'square' method
  bwd_diff(square)(dpx, 1.f);
  printf("d_square at x=4 is %d", dpx.d);
}
```
SLANG.D extends the Slang compiler

- language
- intermediate representation (IR)
- target

1. static analysis
2. generics specialization
3. automatic differentiation

HLSL
GLSL
CUDA
OptiX
SPIR-V
C (CPU)
Automatic differentiation has many *ambiguities*

focus on 3 key issues today

• Which part of the code should we differentiate?

• How do we resolve race conditions in reverse mode AD

• When should we checkpoint vs recompute?
Design principle:
let the user clarify the ambiguity in the language

1. static analysis
2. generics specialization
3. automatic differentiation

... are clarified by the user via language mechanisms
Key idea: use language mechanisms to clarify ambiguities in AD

- Which part of the code should we differentiate?
- How do we resolve race conditions in reverse mode AD?
- When should we checkpoint vs recompute?
Key idea: use language mechanisms to clarify ambiguities in AD

- Which part of the code should we differentiate?
- How do we resolve race conditions in reverse mode AD?
- When should we checkpoint vs recompute?
In practice, we often mix differentiable types/code with non-differentiable ones

```c
struct Foo {
    float x;  
    float y;  
    int z;    
} 
```

want to differentiate w.r.t. x

but not w.r.t. y and z

PyTorch users often have to use `.detach()` to mark code they don’t want to differentiate: a common source of bugs!
In practice, we often mix differentiable types/code with non-differentiable ones.

we need the “differential” structure of Foo, storing derivative information

```c
struct Foo {
    float x;
    float y;
    int z;
};
```

```c
struct DFoo {
    float x;
};
```
In practice, we often mix differentiable types/code with non-differentiable ones.

we need the “differential” structure of Foo, storing derivative information

```
struct Foo {
    struct DFoo {
        float x;
    }
    float y;
    int z;
}
```

How do we let user denote the differential structure?

What if Foo is a generic class?
SLANG.D’s solution:

the **IDifferentiable** interface

similar to the Differentiable protocol in Swift AD

```kotlin
interface IDifferentiable {
    associatedtype D : IDifferentiable
    where D.D==D
    D dzero();
    D dadd(D a, D b);
    D dmul(IScalar s, D d);
}
```

the differential D of a type needs to form a vector space
SLANG.D’s solution:
the IDifferentiable interface
similar to the Differentiable protocol in Swift AD

```d
interface IDifferentiable { 
  associatedtype D : IDifferentiable 
    where D.D==D 
  D dzero(); 
  D dadd(D a, D b); 
  D dmul(IScalar s, D d); 
}

struct Foo : IDifferentiable { 
  typealias D = float; 
  float dzero() {return 0;} 
  // … 

  [DerivativeMember(float)] float x; 
  float y; 
  int z; 
}
```
(can often ask compiler to automatically synthesize these methods)
SLANG.D’s solution: the IDifferentiable interface

similar to the Differentiable protocol in Swift AD

```swift
interface IDifferentiable {
    associatedtype D : IDifferentiable
        where D.D==D
    D dzero();
    D dadd(D a, D b);
    D dmul(IScalar s, D d);
}
```

```swift
interface IMaterial : IDifferentiable {
    ...
}
```

SLANG.D works for modular, polymorphic codebases, without introducing extra overhead
Key idea: use language mechanisms to clarify ambiguities in AD

• Which part of the code should we differentiate?

• How do we resolve race conditions in reverse mode AD?

• When should we checkpoint vs recompute?
Ray tracing is embarrassingly parallel, but its backpropagation is not.

We launch parallel threads for these rays when computing derivatives w.r.t. triangle position, many threads will write into the same memory address.
Each variable can receive vastly different amount of memory writes

computing derivatives w.r.t. position of each triangle

RGB image

# of memory writes (log scale)
Each variable can receive vastly different amount of memory writes

computing derivatives w.r.t. position of each triangle

RGB image  # of memory writes (log scale)

high thread contention, should do parallel reduction

low thread contention, should do atomicAdd

no universal strategy!
Our solution: let users write the derivative accumulation code for global memory buffers

SLANG.D does not provide derivatives of global memory buffer access by default

```
StructuredBuffer<float3> bunny;
RWStructuredBuffer<float3> d_bunny;

[BackwardDerivative(bwd_get_bunny_pos)]
float3 get_bunny_pos(int i) {
    return bunny[i];
}

void bwd_get_bunny_pos(int i, float3 d_pos) {
    atomicAdd(d_bunny[i], d_pos);
}
```

(override the derivative for this function)

(custom derivative)
Our solution: let users write the derivative accumulation code for global memory buffers

SLANG.D does not provide derivatives of global memory buffer access by default

```c++
StructuredBuffer<float3> walls;
RWStructuredBuffer<float3> d_walls;

[BackwardDerivative(bwd_get_walls_pos)]
float3 get_walls_pos(int i) {
    return walls[i];
}

RWStructuredBuffer<float3> buffer;

void bwd_get_walls_pos(int i, float3 d_pos) {
    // accumulate derivatives into a larger buffer
    atomicAdd(buffer[hash(i)], d_pos);
    // in a different pass, accumulate derivatives
    // from buffer to d_walls
}
```
Different derivative accumulation strategies can have significantly different performance tested on a 2D rendering benchmark. Dr. Jit: another AD compiler that opts for a fixed accumulation strategy.
Different derivative accumulation strategies can have significantly different performance tested on a 2D rendering benchmark.

100x between atomicAdd and warp-level reduction in this case.
Key idea: use language mechanisms to clarify ambiguities in AD

• Which part of the code should we differentiate?

• How do we resolve race conditions in reverse mode AD?

• When should we checkpoint vs recompute?
Reverse-mode AD needs to run the loop backwards

```java
x = ...  
for (int i = 0; i < n; i++) {
    y = expensive_func(x);
    z = cheap_func(y);
    x = func(z);
}
// ...
```
Reverse-mode AD needs to run the loop *backwards*

```
x = ... for (int i = 0; i < n; i++) {
    y = expensive_func(x);
    z = cheap_func(y);
    x = func(z);
}
// ...
```

```
dx = ... for (int i = n-1; i >= 0; i--) {
    Z.D dz = bwd_func(z, dx);
    Y.D dy = bwd_cheap_func(y, dz);
    dx += bwd_expensive_func(x, dy);
}
```
Reverse-mode AD needs to run the loop backwards

\[
\begin{align*}
\text{X } x &= \ldots \\
\text{for (int } i = 0; i < n; i++) \{ \\
&\quad \text{Y } y = \text{expensive_func}(x); \\
&\quad \text{Z } z = \text{cheap_func}(y); \\
&\quad x = \text{func}(z); \\
\} \\
\text{// } \ldots
\end{align*}
\]

\[
\begin{align*}
\text{dx } &= \ldots \\
\text{for (int } i = n-1; i >= 0; i--) \{ \\
&\quad \text{Z.D } dz = \text{bwd_func}(z, dx); \\
&\quad \text{Y.D } dy = \text{bwd_cheap_func}(y, dz); \\
&\quad dx += \text{bwd_expensive_func}(x, dy); \\
\}
\]

need access to variables computed in the primal loop
Reverse-mode AD needs to run the loop *backwards*

\[ dx = \ldots \]
\[
\text{for } (\text{int } i = n-1; i >= 0; i--) \{ \\
\quad z.D \ dz = \text{bwd_func}(z, \ dx); \\
\quad y.D \ dy = \text{bwd_cheap_func}(y, \ dz); \\
\quad dx += \text{bwd_expensive_func}(x, \ dy); \\
\}
\]

need to choose whether to **checkpoint** \( x, y, z \) in the primal loop, or to **recompute** them on-the-fly

optimal choice is highly problem dependent, want to give users control
SLANG.D's checkpointing strategy

\[
\begin{align*}
\text{dx} &= \ldots \\
\text{for} \ (\text{int} \ i = n-1; \ i >= 0; \ i--) \ {\{}
\quad \text{Z.D} \ dz = \text{bwd_func}(z, \ dx); \\
\quad \text{Y.D} \ dy = \text{bwd_cheap_func}(y, \ dz); \\
\quad \text{dx} &= \text{bwd_expensive_func}(x_{\text{cached}}[i], \ dy);
\text{\}}
\end{align*}
\]

1. always checkpoint the loop dependent variables
SLANG.D’s checkpointing strategy

1. always checkpoint the loop dependent variables
2. let users decide whether to checkpoint or recompute for each function call

```c
[PreferCheckpoint]  [PreferRecompute]
Y expensive_func(...) {...}  Z cheap_func(...) {...}

dx = ...
for (int i = n-1; i >= 0; i--) {
  Z z = cheap_func(y_cached[i]);
  Z.D dz = bwd_func(z, dx);
  Y.D dy = bwd_cheap_func(y_cached[i], dz);
  dx += bwd_expensive_func(x_cached[i], dy);
}
```
SLANG.D’s checkpointing strategy

1. always checkpoint the loop dependent variables
2. let users decide whether to checkpoint or recompute for each function call
3. recompute everything else

```c

[PreferCheckpoint]
Y expensive_func(...) {...}

[PreferRecompute]
Z cheap_func(...) {...}

dx = ...
for (int i = n-1; i >= 0; i--) {
    Z z = cheap_func(y_cached[i]);
    Z.D dz = bwd_func(z, dx);
    Y.D dy = bwd_cheap_func(y_cached[i], dz);
    dx += bwd_expensive_func(x_cached[i], dy);
}
```

1. always checkpoint the loop dependent variables
2. let users decide whether to checkpoint or recompute for each function call
3. recompute everything else
Checkpointing vs recomputing loop dependent variables in SLANG.D

dx = ...
for (int i = n-1; i >= 0; i--) {
    Z z = cheap_func(y Cached[i]);
    Z.D dz = bwd_func(z, dx);
    Y.D dy = bwd_cheap_func(y Cached[i], dz);
    dx += bwd_expensive_func(x Cached[i], dy);
}

1. always checkpoint the loop dependent variables
2. let users decide whether to checkpoint or recompute for each function call
3. recompute everything else
Checkpointing vs recomputing loop dependent variables in SLANG.D

achieving recomputation using loop unrolling in primal code

\begin{verbatim}
X x = ...  
for (int i = 0; i < n; i+=UNROLL_AMT) {
   [ForceUnroll]
   for (int j = i; j < i+UNROLL_AMT; j++) {
      Y y = expensive_func(x);
      Z z = cheap_func(y);
      x = func(z);
   }
}
// ...  
and mark func as [PreferRecompute]
\end{verbatim}
Checkpointing strategies can have significant performance impact.

*EnzymeCUDA, another AD compiler, crashed at this example at the time.*
SLANG.D applied to a production renderer

> 252,000 lines C++ CPU host code
> 5,000 lines Slang code

30 BOUNCES PATH TRACING

3 BILLION TRIANGLES
3.5M HAIR SEGMENTS
SLANG.D applied to a production renderer

Falcon’s rendering: 68 ms
Our reverse mode AD: 201 ms (2.95x)

modified 200 lines of the original Slang code,
added 100 lines for global buffer accumulation
SLANG.D applied to a rendering model using neural networks

Real-time Neural Appearance Models, Zeltner, Rousselle, Weidlich, Clarberg, Novak, Bitterli, Evans, Davidovic, Kallweit, Lefohn

Training time: **10 minutes** (SLANG.D) vs **4 hours** (PyTorch), SLANG.D is **24x** faster

Single model for training and deployment/inference
SLANG.D used in a PyTorch pipeline

3x code size reduction compared to manual CUDA implementation

*Equal* performance compared to the hand-optimized implementation
SLANG.D comes with a Python interface!

SLANG.D Kernels

```python
> pip install slangpy

import slangpy
import torch

# Generate derivative kernels in CUDA
m = slangpy.loadModule("MyKernel.slang")

# Run kernels & interoperate with PyTorch.
m.render(
    data=torch.rand(100,100),
    ...
).launchRaw(numThreads)
```
Lessons learned from SLANG.D

• We want to reuse the programming model for the forward model (Slang)

• needed to support generics/interfaces as a result

• It is hard for the compiler to automatically synthesize the optimal derivative code

• It is possible to achieve high performance AD by giving users a lot of control
The design space of AD systems

- Tracing-based approach (CppAD/ADOL-C/Adept)
  - often gets tricky in the end and gets abandoned, but very useful to get quick experiments done

- Differentiating general purpose programs (Enzyme/Tapenade)
  - if you have existing code written in the general purpose language, don’t have to change anything!

- Differentiating domain specific programs (Halide/Slang/Taichi)
  - needs to write in many different languages,
    but probably necessary to get the ultimate performance gain